## Design Principle

The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built. Basic design principles enable the software engineer to navigate the design process. Below are the set1 of principles for software design.

1. **The design process should not suffer from “tunnel vision.”** A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job, and the design concepts
2. **The design should be traceable to the analysis model.** Because a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.
3. **The design should not reinvent the wheel.** Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.
4. **The design should “minimize the intellectual distance” between the software and the problem as it exists in the real world.** That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.
5. **The design should exhibit uniformity and integration.** A design is uniform if it appears that one person developed the entire thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.
6. **The design should be structured to accommodate change.** The design concepts enable a design to achieve this principle.
7. **The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.** Well designed software should never “bomb.” It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.
8. **Design is not coding, coding is not design.** Even when detailed procedural designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.
9. **The design should be assessed for quality as it is being created, not after the fact.** A variety of design concepts are available to assist the designer in assessing quality.
10. **The design should be reviewed to minimize conceptual (semantic) errors.** There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. A design team should ensure that major conceptual elements of the design (omissions, ambiguity, and inconsistency) have been addressed before worrying about the syntax of the design model.

When these design principles are properly applied, the software engineer creates a design that exhibits both external and internal quality factors. External quality factors are those properties of the software that can be readily observed by users (e.g., speed, reliability, correctness, usability). Internal quality factors are of importance to software engineers. They lead to a high-quality design from the technical perspective. To achieve internal quality factors, the designer must understand basic design concepts.

### Design Concepts

A set of fundamental software design concepts has evolved over the past decades. Fundamental software design concepts provide the necessary framework for "getting it right."

1. **Abstraction**

When we consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more procedural orientation is taken. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented. Each step in the software process is a refinement in the level of abstraction of the software solution. As we move through the design process, the level of abstraction is reduced. Finally, the lowest level of abstraction is reached when source code is generated. Generally three types of abstractions are used in software design.

A **procedural abstraction** is a named sequence of instructions that has a specific and limited function. An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

A **data abstraction** is a named collection of data that describes a data object. In the context of the procedural abstraction open, we can define a data abstraction called door. Like any data object, the data abstraction for door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).

**Control abstraction** is the third form of abstraction used in software design. Like procedural and data abstraction, control abstraction implies a program control mechanism without specifying internal details. An example of a control abstraction is the synchronization semaphore used to coordinate activities in an operating system.

1. **Architecture**

Software architecture is the overall structure of the software. Architecture is the hierarchical structure of program components (modules), the manner in which these components interact and the structure of data that are used by the components.

1. **Design Patterns**

Design pattern describes a design structure that solves a particular design problem within a specific context.

1. **Modularity**

Modularity is the single attribute of software that allows a program to be intellectually manageable software is divided into separately named and addressable components, often called modules that are integrated to satisfy problem requirements.

1. **Information Hiding**

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

1. **Functional independence**

The concept of functional independence is a direct outgrowth of modularity and the concepts of abstraction and information hiding. Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Independence is measured using two qualitative criteria: cohesion and coupling.

Cohesion is a measure of the relative functional strength of a module. Coupling is a measure of the relative interdependence among modules.

1. **Refinement**

Refinement is actually a process of elaboration. We begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

1. **Refactoring**

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.

When software is re-factored, the existing design is examined for:

* redundancy
* unused design elements
* inefficient or unnecessary algorithms
* poorly constructed or inappropriate data structures
* any other design failure that can be corrected to yield a better design.

1. **Design Classes**

Software team must define a set of design classes. There are five different types of classes; each representing a different layer of architecture.

1. User Interfaces Classes
2. Business Domain Classes
3. Process Classes
4. Persistent Classes
5. System Classes

## Cohesion

Cohesion is a measure of functional strength of a module. A module having high cohesion and low coupling is said to be functionally independent of other modules. By the term functional independence, we mean that a cohesive module performs a single task or function. A functionally independent module has minimal interaction with other modules.

**Classification of cohesion**

The different classes of cohesion that a module may possess are depicted in fig. below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Coincidental** | **Logical** | **Temporal** | **Procedural** | **Communicational** | **Sequential** | **Functional** |

**Low High**

Fig: Classification of cohesion

**Coincidental cohesion:** A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely. In this case, the module contains a random collection of functions. It is likely that the functions have been put in the module out of pure coincidence without any thought or design. For example, in a transaction processing system (TPS), the get-input, print-error, and summarize-members functions are grouped into one module. The grouping does not have any relevance to the structure of the problem.

**Logical cohesion:** A module is said to be logically cohesive, if all elements of the module perform similar operations, e.g. error handling, data input, data output, etc. An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.

**Temporal cohesion:** When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion. The set of functions responsible for initialization, start-up and shutdown of some process, etc. exhibits temporal cohesion.

**Procedural cohesion:** A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective, e.g. the algorithm for decoding a message.

**Communicational cohesion:** A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure, e.g. the set of functions defined on an array or a stack.

**Sequential cohesion:** A module is said to possess sequential cohesion, if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next. For example, in a TPS, the get-input, validate-input, sort-input functions are grouped into one module.

**Functional cohesion:** Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function. For example, a module containing all the functions required to manage employees’ pay-roll exhibits functional cohesion. Suppose a module exhibits functional cohesion and we are asked to describe what the module does, then we would be able to describe it using a single sentence.

## Coupling

Coupling between two modules is a measure of the degree of interdependence or interaction between the two modules. A module having high cohesion and low coupling is said to be functionally independent of other modules. If two modules interchange large amounts of data, then they are highly interdependent. The degree of coupling between two modules depends on their interface complexity.

The interface complexity is basically determined by the number of types of parameters that are interchanged while invoking the functions of the module.

**Classification of Coupling**

Even if there are no techniques to precisely and quantitatively estimate the coupling between two modules, classification of the different types of coupling will help to quantitatively estimate the degree of coupling between two modules.

Five types of coupling can occur between any two modules.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data** | **Stamp** | **Control** | **Common** | **Content** |

**Low High**

**Data coupling:** Two modules are data coupled, if they communicate through a parameter. An example is an elementary data item passed as a parameter between two modules, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for the control purpose.

**Stamp coupling:** Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

**Control coupling:** Control coupling exists between two modules, if data from one module is used to direct the order of instructions execution in another. An example of control coupling is a flag set in one module and tested in another module.

**Common coupling:** Two modules are common coupled, if they share data through some global data items.

**Content coupling:** Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module.

## Functional independence

A module having high cohesion and low coupling is said to be functionally independent of other modules. By the term functional independence, we mean that a cohesive module performs a single task or function. A functionally independent module has minimal interaction with other modules.

**Need for functional independence**

Functional independence is a key to any good design due to the following reasons:

• **Error isolation:** Functional independence reduces error propagation. The reason behind this is that if a module is functionally independent, its degree of interaction with the other modules is less. Therefore, any error existing in a module would not directly effect the other modules.

• **Scope of reuse:** Reuse of a module becomes possible. Because each module does some well-defined and precise function, and the interaction of the module with the other modules is simple and minimal. Therefore, a cohesive module can be easily taken out and reused in a different program.

• **Understandability:** Complexity of the design is reduced, because different modules can be understood in isolation as modules are more or less independent of each other.

## Function-oriented design

The following are the salient features of a typical function-oriented design approach:

1. A system is viewed as something that performs a set of functions. Starting at this high-level view of the system, each function is successively refined into more detailed functions.

For example, consider a function create-newlibrary- member which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge. This function may consist of the following subfunctions:

* assign-membership-number
* create-member-record
* print-bill

Each of these sub-functions may be split into more detailed subfunctions and so on.

1. The system state is centralized and shared among different functions, e.g. data such as member-records is available for reference and updation to several functions such as:
   * create-new-member
   * delete-member
   * update-member-record

## Object-oriented design

In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities). The state is decentralized among the objects and each object manages its own state information.

For example, in a Library Automation Software, each library member may be a separate object with its own data and functions to operate on these data. In fact, the functions defined for one object cannot refer or change data of other objects. Objects have their own internal data which define their state. Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from super class. Conceptually, objects communicate by message passing.

## Function-oriented vs. object-oriented design approach

The following are some of the important differences between function-oriented and object-oriented design.

• Unlike function-oriented design methods, in OOD, the basic abstraction are not real-world functions such as sort, display, track, etc, but realworld entities such as employee, picture, machine, radar system, etc.

For example in OOD, an employee pay-roll software is not developed by designing functions such as update-employee-record, getemployee-address, etc. but by designing objects such as employees, departments, etc. Grady Booch sums up this difference as “identify verbs if you are after procedural design and nouns if you are after object-oriented design”

• In OOD, state information is not represented in a centralized shared memory but is distributed among the objects of the system. For example, while developing an employee pay-roll system, the employee data such as the names of the employees, their code numbers, basic salaries, etc. are usually implemented as global data in a traditional programming system; whereas in an object-oriented system these data are distributed among different employee objects of the system.

Objects communicate by message passing. Therefore, one object may discover the state information of another object by interrogating it. Of course, somewhere or other the real-world functions must be implemented. In OOD, the functions are usually associated with specific real-world entities (objects); they directly access only part of the system state information.

Function-oriented techniques such as SA/SD group functions together if, as a group, they constitute a higher-level function. On the other hand, object-oriented techniques group functions together on the basis of the data they operate on